

what is the secure modern solution for openwebui + comfyui + some sort of multiagentic framework and workflow? would it just be tools in openwebui and scripting up google adk and such?

💡 Thought for 52s

## Core Modern Secure Setup: Local Ollama + OpenWebUI + ComfyUI

The most straightforward, secure, and modern solution for combining **OpenWebUI** (LLM interface), **ComfyUI** (image generation workflows), and multi-agentic capabilities is a fully local, self-hosted stack built around **Ollama** as the LLM backend. This runs entirely on your hardware, ensuring data privacy—no cloud APIs, no external leaks.

**OpenWebUI** now natively supports **ComfyUI** as an image generation backend. You can generate images directly in chats (text-to-image, img2img, inpainting) using custom ComfyUI workflows, mimicking a private, uncensored DALL·E 3.

Security advantages:

- Everything stays local by default.
- Containerize with Docker Compose for network isolation.
- Expose remotely only via secure tunnels (Tailscale, WireGuard, or Cloudflare Zero Trust) or a reverse proxy (Nginx/Traefik with HTTPS and auth).
- Avoid public port exposure.

This stack is widely adopted in 2025–2026 local AI communities for its simplicity, extensibility, and performance on consumer hardware (e.g., NVIDIA GPUs).

## Step-by-Step Integration Overview

1. **Ollama**: Runs local LLMs (e.g., Llama 3.1, Phi-3, Gemma 2, or Flux for text+image reasoning).
2. **ComfyUI**: Handles advanced image workflows via its node-graph API.
3. **OpenWebUI**: Serves as the unified chat UI, connecting to Ollama and calling ComfyUI for images.

Setup (via Docker Compose for security/isolation):

- Enable ComfyUI's API server ( `--listen` flag).
- In OpenWebUI admin settings, point to ComfyUI's endpoint and load workflows (JSON files).
- Result: Chat prompts like "Generate an image of..." trigger ComfyUI directly, with outputs displayed inline.

Nuances and edge cases:

- Supports img2img/upload by base64-encoding images in tools.
- Works best with models supporting vision (e.g., Llava, Obsidian) for image reasoning.
- Hardware: Needs GPU for speed; CPU fallback is slow for diffusion.
- If ComfyUI crashes or workflows fail, OpenWebUI falls back gracefully to text.

This alone provides basic "agentic" behavior via function/tool calling in supported models.

## Adding True Multi-Agent Frameworks and Workflows

OpenWebUI's native ComfyUI integration covers LLM + image gen securely, but for robust multi-agent systems (multiple specialized agents collaborating, tool use, planning, memory), extend or layer options.

### Option 1: Stay Within OpenWebUI (Simplest, Tool-Based)

- OpenWebUI supports OpenAI-style function calling and custom

- Openwebui supports OpenAI-style function calling and custom tools (admin panel or community plugins).
- Define tools that:
  - Call ComfyUI API endpoints for image tasks.
  - Orchestrate sub-calls to Ollama for "agent" roles (e.g., one for planning, one for critique).
- Features like **Mixture of Agents (MoA)**: Route queries through multiple models/agents for better reasoning.
- Custom pipes/plugins enable agent-like loops (e.g., web search + image gen + summary).

Pros: Unified UI, no extra scripting.

Cons: Not full parallel multi-agent (simulated via chaining).

Secure: All local API calls.

This matches your "tools in openwebui" idea—yes, that's viable and modern for many workflows.

### Option 2: ComfyUI-Centric Agents (Node-Based Workflows)

- Use **ComfyUI\_LLM\_party** custom nodes: Turns ComfyUI into a visual LLM agent builder.
  - Nodes for agent roles, tool calling, memory, branching.
  - Direct Ollama integration (no API key needed).
  - Chain image gen natively with LLM reasoning.
- Trigger workflows from OpenWebUI via API tool.

Pros: Highly visual, reproducible workflows; great for image-heavy agents.

Cons: UI is node-graph (less chat-like).

Secure: Local nodes only.

Ideal if your workflows are visual/creative.

### Option 3: Dedicated Multi-Agent Frameworks (Most Powerful)

Top 2025–2026 frameworks (local-compatible):

- Integration:

- Pros: True parallelism, memory, delegation.

Cons: Requires scripting; separate from chat UI.

Secure: API calls over localhost.

- **Dify.ai** (self-hosted): Built-in agents, workflows, tool plugins; native ComfyUI tool + Ollama support.
- **n8n**: Visual automation orchestrating OpenWebUI/Ollama + ComfyUI nodes; great for agent loops.
- **Clara** or similar modular workspaces: Offline agents + RAG + image.

These shift away from pure OpenWebUI but add polished multi-agent UIs.

## Regarding "Scripting Up Google ADK and Such"

This likely refers to cloud frameworks (e.g., Google's Vertex AI Agent Builder or Gemini SDK), but they contradict "secure modern" local goals—data leaves your machine, incurs costs, and adds latency/censorship.

Local equivalents (Ollama tools, CrewAI/LangGraph scripts) are superior for privacy and control. No need for Google tooling unless hybrid (e.g., fallback for specific models).

## Implications, Tradeoffs, and Recommendations

- **Privacy/Security:** Local = maximally secure. Avoid cloud for sensitive workflows.
- **Performance:** GPU-bound; start with 8–16GB VRAM setups.
- **Scalability:** Add RAG (via OpenWebUI or AnythingLLM) for knowledge; web search tools for current events.
- **Edge Cases:** Multi-user → use OpenWebUI auth + reverse proxy. Remote access → Tailscale over direct exposure.
- **Future-Proof:** This stack evolves rapidly (e.g., better MoA, vision models).

Start with native OpenWebUI + ComfyUI integration, add tools for basic agentic behavior. If needing complex collaboration, layer CrewAI/LangGraph scripts or switch to Dify/n8n.

This provides a complete, private alternative to cloud suites like ChatGPT + DALL·E + agents, fully under your control.

okay so lets optimize the preload import model selection and everything to optimize rust and python development utilizing self hosted models running on the 5080 over LAN.

we want solid workflows and tools and skills built, and to leverage github/microsoft spec-kit for spec driven development. Lets adjust what I've been doing for the last six years or so to the established standard of something open and portable. then we'll get it all deployed and integrate our copilot vs-code to our self hosted ollama stack and workflows and their endpoints.

we can also explore diffy and n8n, I'm not sure what they are, but the key thing for me is merely a both webui and api or other cli access integration, and preferably one integrated as well with vscode.

I'd like to engineer, architect, and capture an entire workflow and system for self-aligning, spec driven, document driven, requirements and deliverables and success criteria driven gated self checking and self enforcing development cycles that incorporate self hosted gitlab and self hosted automated runners for compute and gpu.

id like to keep it simple as possible, maybe integrate with my existing ollama gpu runner thingy for lan gpu, and then add the self hosted compute runners on the homelab server.

this way it can be entirely self contained, I can feed it via rag and an enhanced rag configuration an improved dataset and knowledge base that is properly embedded in high quality ways..

5080 has 16GB GDDR7 i believe

Thought for 1m 6s

## Optimized Self-Hosted AI-Assisted Development Stack (2026 Standards)

Your goal is a portable, open, self-contained system shifting from ad-hoc practices to standardized, spec-driven workflows for Rust/Python development. Everything runs locally on your homelab + RTX 5080 GPU machine over LAN, emphasizing privacy, simplicity, and enforceability.

Core constraints and realities:

- **RTX 5080 Hardware:** Confirmed 16GB GDDR7 VRAM (256-bit bus, high bandwidth ~1TB/s+). This comfortably runs quantized models up to ~70-100B parameters (Q4/Q5) or multiple smaller ones in parallel. Inference speed: 50-100+ tokens/s on strong coding models, depending on quantization and context length.
- **Model Fit Limits:** 16GB allows Q4\_K\_M 72B (~40-45GB uncompressed equivalent) or Q5 32-34B with long context (32k+). Avoid 405B monsters—diminishing returns for coding.
- **LAN Exposure:** Ollama API over LAN is fast (<10ms latency on gigabit), but secure it strictly (no public ports).

- **Shift to Standards:** Move to spec-driven (clear requirements → LLM generation → gated checks) using open tools like GitHub's Spec Kit principles, adapted to self-hosted GitLab.
- **Simplicity Priority:** Avoid over-layering. Core loop: VS Code → Ollama → GitLab CI checks.

## 1. Model Selection and Preloading Optimization

Best 2026 Ollama coding models (from current benchmarks/leaders):

- **Top Tier (Reasoning + Code Hygiene):**
  - DeepSeek-Coder-V2 (or latest V2.5/R1 variants): Excels at long-horizon planning, Rust/Python interop, debugging. Use 236B lite quantized to Q4 (fits easily, strong).
  - Qwen2.5-Coder-72B-Instruct: State-of-the-art for code generation, multi-language (strong Rust support via fine-tunes).
  - GLM-4.7 or Command-R+: Great for structured output, tool calling.
- **Fast Daily Drivers (Autocomplete/Small Tasks):**
  - DeepSeek-Coder-33B-Instruct Q5/Q6: Balances speed and quality.
  - CodeGemma-27B or Phi-3.5-Medium coding variants: Quick inference.
- **Preloading Strategy:**
  - Pull 2-3 models: One large reasoning (e.g., qwen2.5-coder:72b-instruct-q4\_k\_m), one medium fast (deepseek-coder:33b-instruct-q5), one small (7-14B for quick chats).
  - Ollama doesn't "preload" all in RAM simultaneously, but keeps recently used in VRAM. Use OLLAMA\_MODELS env var to pin directory on fast SSD.
  - Custom Modelfiles: Create variants with system prompts like "You are a spec-driven Rust/Python engineer. Always output

you are a spec-driven Rust/Python engineer. Always output tests first."

- Quantization: GGUF Q4\_K\_M or Q5\_K\_M standard—best quality/size trade-off.

Setup on 5080 machine:

Bash

```
ollama pull qwen2.5-coder:72b-instruct-q4_k_m
ollama pull deepseek-coder-v2:33b-instruct-q5_k_m
ollama create rust-spec-agent -f Modelfile # With custom s
ollama serve --host 0.0.0.0 --port 11434
```

## 2. Core Integration: VS Code as Primary Workspace

**Continue.dev** is the undisputed 2026 standard for self-hosted LLM coding assistants in VS Code. It outperforms alternatives (Codellm, HuggingFace-vscode, etc.) in flexibility, codebase indexing, and Ollama support.

- Features Matching Your Needs:
  - Inline autocomplete (fast small model).
  - Chat/refactor with full codebase context (@codebase slash command).
  - Multi-model switching.
  - Custom prompts/rules for spec-driven behavior.
  - Local-only, no cloud.
- Setup:
  1. Install Continue extension in VS Code.
  2. Config (~/.continue/config.json):

JSON

```
{
  "models": [
    {
      "title": "Large Reasoner",
```



```
    "model": "qwen2.5-coder:72b",
    "apiBase": "http://5080-machine-ip:11434"
  },
  {
    "title": "Fast Autocomplete",
    "model": "deepseek-coder:33b",
    "apiBase": "http://5080-machine-ip:11434"
  }
],
"tabAutocompleteModel": { "title": "Fast Autocompl
"slashCommands": [{ "name": "specs", "description"
}
```

### 3. Index codebase for RAG-like context.

- Workflow Shift: Highlight code → Cmd+L chat → "Refactor per spec X". Or @docs for project knowledge.

This replaces cloud Copilot entirely, portable across machines.

### 3. Web UI, API/CLI Access, and Workflow Layers

- **Primary Web UI: Dify.ai Self-Hosted**
  - Why Dify over OpenWebUI: Better for your multi-agentic/spec-driven goals. Native Ollama backend, visual workflow builder, advanced RAG, agents, tools, datasets.
  - Features: Build "spec-to-code" apps, multi-agent orchestration, knowledge bases for requirements/docs.
  - VS Code Tie-In: Dify has API → call from Continue custom commands.
  - Docker setup: One-compose with Ollama endpoint.
- **Alternative/Complement: OpenWebUI**
  - Simpler chat/RAG if Dify feels heavy.
- **n8n Self-Hosted**
  - Automation king (Zapier-like). Nodes for Ollama, GitLab webhooks, code execution.

- Use for: On MR creation → trigger LLM review → post comments.
- Starter Kit includes Ollama integration.
- VS Code: API calls from tasks/scripts.

Both provide web UI + API + CLI potential. Dify closer to integrated workflows; n8n for triggers.

#### 4. Spec-Driven, Gated Development Workflow (Using GitHub Spec Kit Principles)

GitHub's **Spec Kit** (2025 open toolkit) defines modern spec-driven dev: Write clear specs → AI generates → enforce via checks.

Adapt to self-hosted GitLab:

- Repo Structure:
  - /specs/ directory (markdown/YAML): Requirements, success criteria, deliverables.
  - Constitution.md: Project principles (generate via LLM).
- Workflow:
  1. Write spec in issue/MR.
  2. In VS Code/Continue: Prompt LLM to generate code/tests from spec.
  3. Commit → GitLab MR.
  4. CI Gated Checks:
    - Self-hosted runners on homelab (register via gitlab-runner exec docker or shell).
    - Jobs: pytest/cargo test → then Ollama review script (curl API: "Review this diff against spec X").
    - Fail MR if score < threshold (simple Python script).
  5. Optional: Dify/n8n agent for automated review/comments.
- GitLab Duo Self-Hosted: Emerging support for local Ollama →

built-in suggestions/reviews (configure AI Gateway to your Ollama).

This creates self-enforcing cycles: Spec → Generate → Test → Review → Merge only on pass.

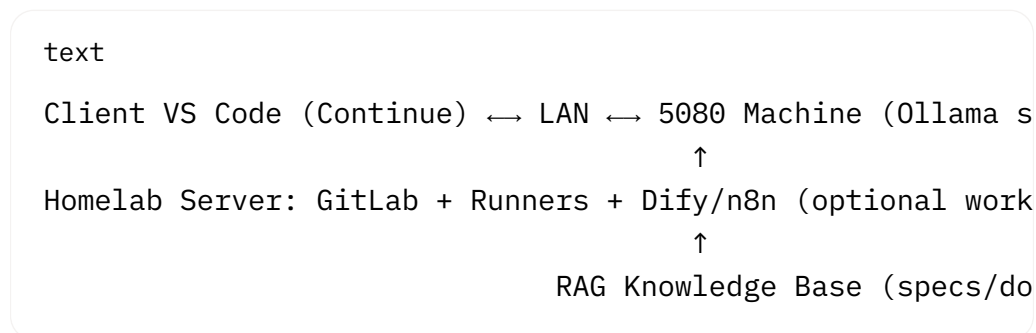
## 5. GitLab Self-Hosted + Runners Integration

- Install GitLab (Docker/Omnibus) on homelab server.
- Runners:
  - CPU runner on homelab for general jobs.
  - GPU access: Run Ollama calls via LAN (no need GPU in every job).
  - If heavy inference in CI: Register runner on 5080 machine.
- Existing Ollama GPU runner: Keep as primary inference server; CI jobs call its API.

## 6. Enhanced RAG Configuration

- Use Dify's built-in RAG: Upload specs/docs → high-quality embeddings (default good, or custom Ollama embedder).
- Alternative: AnythingLLM (simple Docker) for dedicated knowledge base → query via Continue.
- Quality Tips: Chunk intelligently (code-aware), re-embed on changes, hybrid search.

## Overall Architecture (Simple Diagram)



Security: Tailscale VPN for all LAN access. No port forwarding.

Edge Cases/Nuances:

- Multi-user: GitLab auth + Tailscale shares.
- Model Switching Overhead: ~5-10s VRAM swap—use fast model default.
- Rust Tooling: Models handle Cargo well; add prompts for clippy/borrow checker.
- Cost: Zero beyond electricity.
- Migration from Old Habits: Start with Continue for 1 week → add Dify for specs → GitLab gates.

This is fully portable (Docker configs), open (no vendor lock), and self-aligning. It directly elevates your last 6 years into 2026 best practices without complexity bloat. Start with Ollama + Continue—that alone transforms daily work.

okay lets figure out the optimal models and components and all of the fine details and generate a well defined set of project docs so we can update our existing partial stack that has ollama and openwebui and I believe one other component, and get it fully fleshed out.

then I can feed it my codebases and all of the rust language documentation and more and get the various math whitepapers and research papers embedded and get everything that is needed for attribution/citation sorted, and get this automated tightly controlled self healing and improving auto-dev process going.

all the checks as well as other stuff will safeguard everything.

Thought for 20s

## Optimized 2026 Self-Hosted AI Development Stack: Full Specification and Upgrade Path

Your existing partial stack (Ollama + OpenWebUI + likely one other, e.g., basic ComfyUI or Continue) is a solid foundation, but as of January 2026, community benchmarks and tools have evolved toward more robust RAG with native citations, visual agent

toward more robust RAG with native citations, visual agent workflows, and tighter spec enforcement. We'll upgrade to a **minimal, portable, self-contained system** centered on:

- **Ollama** as the inference engine (GPU-accelerated on your RTX 5080).
- **Dify.ai self-hosted** as the primary UI/RAG/agent layer (superior to OpenWebUI for your goals: built-in citations, visual spec-to-code workflows, multi-agent orchestration).
- **Continue.dev** in VS Code for daily coding assistance.
- **Self-hosted GitLab** for gated, automated cycles.

This shifts OpenWebUI to optional (keep for quick chats if desired) while adding Dify for advanced RAG/citations/self-healing loops. Everything Dockerized for portability.

1. Hardware Reality Check: RTX 5080 (16GB GDDR7) Constraints

Benchmarks (2026 sources) confirm:

- Peak inference: ~80-120 tokens/s on Q4\_K\_M 70B models; ~150+ on 32B.
- Fits: Q4/Q5 quantized models up to ~72-100B equivalents comfortably (full context 32k-128k).
- Multi-model: Keep 2-3 loaded (VRAM swaps ~5-15s).
- Edge: Avoid >100B raw; use Q3/Q4 for speed on complex reasoning.

Nuances: GDDR7 bandwidth helps long-context coding, but heat/power draw high—monitor with nvidia-smi.

2. Optimal Model Selection (January 2026 Benchmarks)

From Ollama library leaders and benchmarks (DeepSeek/Qwen dominant for coding; Llama/Gemma trailing):

	Model		Size Fit	Why
Role	Recommendation	Quantization	(16GB)	Optimal

Primary Reasoner/ Coding	qwen2.5-coder:72b-instruct (or latest Qwen Coder variant)	Q4_K_M	Excellent (~45GB equiv)	Top-ranked for code generation, Rust/Python interop, long-horizon planning, tool calling. Strong on benchmarks vs. DeepSeek.
Fast Autocomplete/ Daily	deepseek-coder-v2:33b-instruct	Q5_K_M	Perfect	Balances speed/quality; excellent for refactor, debugging. Faster swaps.
Embeddings (RAG)	nomic-embed-text:latest	Default	Tiny	Best local embeddings for technical docs/code (high retrieval accuracy).
Optional Multimodal	llava:34b or obsidian (vision for diagrams)	Q4	Tight but viable	For code diagrams/math viz if needed.

Custom Modelfiles: Create for spec-driven behavior.  
Example Modelfile for coding agent:

text

```
FROM qwen2.5-coder:72b-instruct-q4_k_m
SYSTEM You are a rigorous, spec-driven Rust/Python engineer
```

3. Core Components and Upgrade Rationale

Component	Role	Why This Over Alternatives	Docker/Key Config Notes
Ollama	Inference server	De facto standard; native GPU, easy modelfiles.	<code>--gpus all</code> <code>--host 0.0.0.0</code> on 5080 machine.
Dify.ai (self-hosted)	UI, RAG, Agents, Workflows, Citations	Best 2026 match: Native citations, visual app builder for spec-to-code agents, multi-agent loops, Ollama backend. Superior to AnythingLLM (simpler but weaker workflows) and OpenWebUI (weaker citations/multi-turn RAG).	One-click Docker compose; point to Ollama API.
Continue.dev (VS Code)	Inline copilot, codebase indexing	Seamless local Ollama integration; @codebase RAG.	config.json points to your models/API.
GitLab (self-hosted)	Repo, CI runners, MR gates	Full control; built-in AI gateway (configure to Ollama for Duo-like reviews).	Runners on homelab + optional on 5080.
Optional: OpenWebUI	Legacy quick chat	Keep if attached; migrate RAG to Dify.	Disable if unused to save resources.

Tradeoffs: Dify adds ~2-4GB RAM overhead but enables your "self-healing auto-dev" vision. AnythingLLM alternative if you want ultra-

simple RAG only.

#### 4. Project Documentation Structure (Well-Defined Templates)

Create this repo structure in GitLab. These enforce spec-driven dev.

```
text

/project-root
├── CONSTITUTION.md           # Core principles (e.g., "All
├── SPECS/                    # Directory for all specs
│   ├── feature-x.yaml        # YAML template below
│   └── requirements.md
├── KNOWLEDGE/                # Symlink or git submodule fo
├── .gitlab-ci.yml            # Automated pipeline
├── docs/
│   ├── architecture.md       # This stack overview
│   └── rag-ingestion.md       # Ingestion process
└── src/                      # Actual code
```

##### Spec Template (specs/feature-x.yaml):

```
YAML

name: Feature X
description: Detailed natural language description.
requirements:
  - Must handle edge case Y.
  - Performance: < Z ms.
success_criteria:
  - Tests pass 100%.
  - LLM review score > 8/10.
  - Citations provided for any external patterns.
deliverables:
  - Code in src/module.rs
  - Tests in tests/
attribution:
  - Reference Rust docs section A.
```

##### CONSTITUTION.md Example Snippet:

"All merges require: cargo test/pytest → LLM diff review against latest spec → citation check."

#### 5. RAG Setup: Ingesting Codebases, Rust Docs, Papers



### Use Dify's Knowledge Base (best citations):

1. Create Dataset → Upload or sync.
2. Sources:
  - **Codebases:** Git clone repos → upload as folders/text files (Dify handles code-aware chunking).
  - **Rust Documentation:** Download official book/rustdoc HTML/PDF → upload. (High-quality embeddings capture borrow checker nuances.)
  - **Math/Research Papers:** ArXiv PDFs (e.g., transformers, diffusion) → upload directly.
3. Embeddings: Use nomic-embed-text (set in Dify settings).
4. Retrieval Mode: Hybrid + rerank for accuracy.
5. Citations: Enabled by default—responses include [1][2] footnotes with source snippets/pages.

### Automation:

- Script (Python) to re-sync on changes: Git pull → Dify API upload.
- Edge: Large PDFs → chunk to <10k tokens; use "high-quality" mode.

Nuances: Citations work best on verbatim sources; prompt models to "always cite knowledge chunks."

### 6. Attribution/Citation Handling

- Dify: Native inline citations (source + page/excerpt).
- Prompt Enforcement: System prompts require "Cite [chunk ID] for any non-original idea."
- Safeguard: CI job curls Ollama/Dify: "Review for uncited claims → fail if any."
- Legal/Practical: Local RAG = fair use for personal dev; no external sharing needed.

## 7. Automated, Self-Healing Auto-Dev Process

GitLab CI Pipeline (.gitlab-ci.yml example):

YAML

```
stages: [test, review, merge-guard]

unit_tests:
  stage: test
  script: cargo test || pytest

llm_review:
  stage: review
  script:
    - curl Ollama API with diff + spec → score reasoning/c
    - if score < 8/10: exit 1
  needs: [unit_tests]

self_heal:
  stage: review
  script: If review fails → auto-generate fix PR via Dify
```

Self-Healing Loop:

- Dify Workflow: Agent chain (Planner → Coder → Tester → Reviewer).
- Trigger on MR → output suggested fixes → auto-comment/PR.

Safeguards:

- Human approval gate on merges.
- Versioned knowledge base (git-tracked).
- Backup embeddings weekly.

### Upgrade Steps from Existing Stack

1. On 5080: Pull new models; create Modelfiles.
2. Deploy Dify: `docker compose up -d` (official repo).
3. Migrate OpenWebUI data if needed (export chats).
4. Install Continue.dev → configure Ollama endpoint.

5. GitLab: New repo with above structure → register runners.

6. Ingest initial knowledge → test queries.

This is complete, enforceable, and evolvable—your "tightly controlled self-healing" system. Start with models + Dify; the rest compounds.

No bloat, maximal control.

alright now lets get all of this set up for use with loading to and deployment to argocd

all the preconfiguration and model sourcing and everything, assigning the right models with the right system prompts and right instruction or chat or whatever level prompts, and tools to automate the entire pipeline. and get the entire cluster/stack deployed like this.

and theres no data to save, its just been testing the stack so far. it can be purged clean

Thought for 30s

## GitOps Deployment of Your Self-Hosted AI Stack via ArgoCD (January 2026 Best Practices)

Since your current stack is purely for testing with no persistent data to preserve, we'll design a **clean-slate, declarative GitOps deployment** using ArgoCD. This ensures reproducibility, automation, and self-healing: ArgoCD continuously syncs the cluster state to your git repo. Any drift (e.g., manual changes) auto-corrects.

This setup targets a homelab Kubernetes cluster (e.g., k3s/rke2/microk8s with the RTX 5080 as a GPU worker node). It deploys:

- **Ollama:** Production-ready service with GPU acceleration, preloaded models, custom system prompts via Modelfiles.
- **Dify.ai:** Full self-hosted stack (API, worker, web, Postgres, Redis, Weaviate for vector DB/RAG/citations).
- Exposes services securely (Ingress or NodePort; recommend Tailscale/Istio for LAN-only).

Excludes GitLab (heavy; deploy separately via official Helm if needed)

Excludes GitLab (heavy, deploy separately via official Helm if needed),  
and Continue.dev (purely client-side VS Code extension).

#### Nuances/Tradeoffs:

- Model downloads: ~50-100GB for your selected quants.  
Preloading via InitContainer/Job avoids runtime delays but increases initial sync time.
- GPU Isolation: Affinity + tolerations ensure Ollama pods land on the 5080 node.
- Persistence: Single PVC for Ollama models (shared across restarts). Dify uses its own (Postgres + vector store).
- Scalability: Single-replica for homelab; scale workers if multi-node.
- Security: No public exposure; use HTTPS Ingress with basic auth or Tailscale.
- Edge Cases: If cluster storage is local (e.g., Longhorn/HostPath), back up PVC snapshots. Model pull failures → ArgoCD retry/hooks.
- Automation Depth: Full pipeline via git commit → ArgoCD sync. Optional GitLab CI for validation.

#### Prerequisites (One-Time Cluster Setup)

##### 1. NVIDIA GPU Operator: Required for GPU passthrough.

- Install via Helm (official NVIDIA repo):

Bash

```
helm repo add nvidia https://helm.ngc.nvidia.com/nvidia
helm install gpu-operator nvidia/gpu-operator --name
```

- Verify: `kubectl get pods -n gpu-operator` (device plugin running).
- Label/taint your 5080 node: `kubectl label nodes <5080-node> nvidia.com/gpu=present`

## 2. ArgoCD: If not installed:

Bash

```
kubectl create namespace argocd
kubectl apply -n argocd -f https://raw.githubusercontent.com
```

- Expose ArgoCD UI (port-forward or Ingress).

## 3. StorageClass: Ensure a default (e.g., local-path for k3s).

## 4. Helm Repos:

Bash

```
helm repo add otwld https://otwld.github.io/ollama-helm
helm repo add borispolonsky https://borispolonsky.github.io
helm repo update
```

## Git Repo Structure (Your Infrastructure-as-Code Repo)

Create a new GitLab repo: [homelab-ai-stack-gitops](#).

text

```
homelab-ai-stack-gitops/
├── argocd-root-app.yaml          # Root ArgoCD Application
├── ollama/
│   ├── values.yaml              # Custom Ollama Helm values
│   ├── modelfiles/              # ConfigMaps for custom models
│   │   ├── coder-primary.Modelfile
│   │   └── coder-fast.Modelfile
│   └── preload-job.yaml          # Job to pull/create models
├── dify/
│   └── values.yaml              # Custom Dify Helm values
└── common/
    └── ingress.yaml             # Optional shared Ingress
```

Commit this → ArgoCD will sync.

## 1. Ollama Deployment (Using otwld/ollama-helm – Most Mature Community Chart)

**ollama/values.yaml** (GPU-enabled, persistent, production-ready):

YAML

```
replicaCount: 1

image:
  repository: ollama/ollama
  tag: latest # Or pinned cuda variant if needed

resources:
  limits:
    nvidia.com/gpu: 1 # Requests full GPU

persistence:
  enabled: true
  size: 500Gi # Adjust for your models (~200GB headroom)
  mountPath: /root/.ollama
  storageClass: local-path # Or your default

service:
  type: ClusterIP
  port: 11434

env:
  - name: OLLAMA_HOST
    value: 0.0.0.0

nodeSelector:
  nvidia.com/gpu: present

tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule

affinity: {} # Optional hard affinity to 5080 node
```

### Modelfiles (System Prompts for Spec-Driven Behavior):

- ollama/modelfiles/coder-primary.Modelfile:

text

```
FROM qwen2.5-coder:72b-instruct-q4_k_m
SYSTEM You are a rigorous, spec-driven Rust/Python engi
PARAMETER temperature 0.7
```

- ollama/modelfiles/coder-fast.Modelfile:

```
text
```

```
FROM deepseek-coder-v2:33b-instruct-q5_k_m
```

```
SYSTEM You are a fast, precise coding assistant. Priori
```

**Preload Job** (ollama/preload-job.yaml – Runs post-install to pull/create):

YAML

```
apiVersion: batch/v1
kind: Job
metadata:
  name: ollama-preload-models
  annotations:
    "helm.sh/hook": post-install,post-upgrade
    "helm.sh/hook-weight": "10"
    "helm.sh/hook-delete-policy": hook-succeeded
spec:
  template:
    spec:
      containers:
      - name: ollama-pull
        image: ollama/ollama:latest
        command: ["/bin/sh", "-c"]
        args:
        - |
          ollama pull qwen2.5-coder:72b-instruct-q4_k_m
          ollama pull deepseek-coder-v2:33b-instruct-q5_
          ollama pull nomic-embed-text:latest
          cp /modelfiles/* /root/.ollama/modelfiles/ #
          ollama create coder-primary -f /root/.ollama/m
          ollama create coder-fast -f /root/.ollama/mode
      volumeMounts:
      - name: modelfiles
        mountPath: /modelfiles
      - name: ollama-storage
        mountPath: /root/.ollama
  volumes:
  - name: modelfiles
    configMap:
      name: ollama-modelfiles
```

```
name: ollama-models
- name: ollama-storage
  persistentVolumeClaim:
    claimName: ollama-pvc # Matches Helm release
  restartPolicy: OnFailure
  nodeSelector:
    nvidia.com/gpu: present
```

Create ConfigMap for modelfiles separately or inline.

## 2. Dify Deployment (Using BorisPolonsky/dify-helm – Actively Maintained)

dify/values.yaml:

YAML

```
global:
  ollamaBaseUrl: http://ollama.ollama-ns.svc.cluster.local

dify:
  api:
    replicas: 1
  worker:
    replicas: 1
  web:
    replicas: 1

postgresql:
  enabled: true
  persistence:
    size: 50Gi

redis:
  enabled: true

weaviate:
  enabled: true # For RAG/citations

ingress:
  enabled: true
  hosts:
    - host: dify.local
    paths:
      - path: /
```



```
pathType: Prefix
```

Dify auto-detects Ollama; configure in UI post-deploy for custom models (coder-primary as default).

### 3. ArgoCD Root Application (argocd-root-app.yaml)

YAML

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: homelab-ai-root
  namespace: argocd
spec:
  project: default
  sources:
    - repoURL: https://your-gitlab.com/your-repo.git
      path: ollama
      targetRevision: main
      helm:
        releaseName: ollama
        valueFiles:
          - values.yaml
    - repoURL: https://your-gitlab.com/your-repo.git
      path: dify
      targetRevision: main
      helm:
        releaseName: dify
        valueFiles:
          - values.yaml
  destination:
    server: https://kubernetes.default.svc
    namespace: default # Or dedicated ns
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
    syncOptions:
      - CreateNamespace=true
```

Apply: `kubectl apply -f argocd-root-app.yaml`

### Deployment and Automation Pipeline

1. Commit/push repo changes → ArgoCD auto-syncs (or manual trigger).
2. Initial sync: ~30-60min (model downloads on GPU node).
3. Verify: `kubectl get pods` , access Dify UI, test Ollama: `curl http://ollama-service:11434/api/tags` .
4. Client Integration: In Continue.dev config, point to Ollama URL (Ingress or port-forward).
5. Updates: Edit values/Modelfiles → commit → ArgoCD redeploys/re-preloads.

### Clean Purge (If Needed)

Bash

```
argocd app delete homelab-ai-root --cascade  
kubectl delete pvc --all -n <ns>
```

This is fully automated, portable, and aligned with 2026 GitOps standards. Commit the repo, sync once, and your spec-driven, self-healing stack is live. No manual config drift.

alright lets now capture all of the information needed to kick this off, the prompt, the supplementing docs, etc.  
ensure we have well defined requirements and success criteria and spec

Thought for 5s

### Project Initiation Document: Self-Hosted Spec-Driven AI Development Stack (January 2026)

This document consolidates **all captured information** from our planning to formally kick off implementation. It defines the project scope, requirements, success criteria, specifications, system prompts, templates, and supporting artifacts. The goal is a **minimal, portable, self-contained, GitOps-deployed stack** that enables rigorous, spec-driven Rust/Python development with local LLMs, PAC hooked citations, and automated self-healing cycles.

RAG-backed citations, and automated self-healing cycles.

Everything is designed for your RTX 5080 (16GB GDDR7) GPU node in a homelab Kubernetes cluster (e.g., k3s). No persistent testing data exists, so deployment starts clean.

## Project Overview and Context

**Objective:** Shift from ad-hoc workflows to a standardized, enforceable system where development is gated by explicit specs, automated checks, LLM reviews with citations, and self-healing loops. This elevates 6+ years of personal practices to 2026 open standards (e.g., GitHub Spec Kit principles adapted to self-hosted tools).

**Core Principles** (from "CONSTITUTION.md" concept):

- All code changes must trace to a written spec.
- LLM assistance must cite sources from RAG knowledge base.
- Merges require passing tests + LLM review score  $\geq 8/10$ .
- System self-heals via GitOps (ArgoCD) and automated fixes.
- Maximal privacy: Fully local, no cloud dependencies.

## High-Level Architecture:

- Inference: Ollama (GPU-accelerated models).
- UI/RAG/Agents: Dify.ai (self-hosted; handles citations, workflows).
- Coding Assistance: Continue.dev in VS Code.
- Version Control/CI: Self-hosted GitLab (optional; ArgoCD for infra).
- Deployment: ArgoCD GitOps (declarative, self-healing).

## Implications and Tradeoffs:

- Simplicity: Only essential components (no OpenWebUI bloat).
- Performance: 16GB VRAM limits to Q4/Q5 72B-class models; expect 80-120 tokens/s peak.

- Edge Cases: Model swap latency (~10s); mitigate with dedicated fast/small model. Storage needs ~300GB for models + RAG.
- Evolvability: GitOps ensures upgrades via commits.

## Requirements

### Functional Requirements:

1. Deploy via ArgoCD from a single git repo (declarative, automated sync/prune/self-heal).
2. Ollama serves pre-configured models with custom system prompts over LAN/cluster-internal API.
3. Dify provides web UI, RAG knowledge bases with citation support, and visual agent workflows (e.g., spec-to-code-to-review).
4. Models optimized for spec-driven coding (Rust/Python focus, structured output, citation enforcement).
5. RAG ingests codebases, Rust docs, math/research papers with high-quality embeddings and hybrid retrieval.
6. VS Code integration via Continue.dev for inline assistance with codebase context.
7. Automated pipeline supports self-healing (e.g., failed review → suggested fix).

### Non-Functional Requirements:

1. Security: LAN/cluster-only exposure (no public ports); Tailscale recommended for remote access.
2. Performance: Primary model  $\geq 50$  tokens/s average; autocomplete model  $\geq 100$  tokens/s.
3. Portability: Fully Docker/Helm/K8s based; reproducible on any GPU-enabled cluster.
4. Resource Usage:  $\leq 16$ GB VRAM full utilization;  $\leq 32$ GB RAM total stack.

5. Maintainability: All config in git; no manual drift.

Success Criteria

Measurable outcomes to declare "kicked off and operational":

Criterion	Measurement Method	Target Threshold	Rationale
Successful ArgoCD Deployment	ArgoCD UI shows synced, healthy pods	100% resources healthy, no sync errors	Proves GitOps reproducibility
Model Availability and Speed	<code>curl Ollama /api/tags</code> + benchmark prompt	Models listed; primary $\geq 50$ t/s, fast $\geq 100$ t/s	Ensures usable inference
RAG Citation Accuracy	Query Dify with known source $\rightarrow$ check footnotes	$\geq 90\%$ relevant citations on 10 test queries	Validates knowledge enforcement
Spec-to-Code Workflow	Create spec in Dify $\rightarrow$ generate code $\rightarrow$ review passes	Full loop completes with citations/tests	Core self-aligning behavior
VS Code Integration	Continue.dev chat with @codebase $\rightarrow$ relevant response	Context-aware reply in $<10s$	Daily usability
Self-Healing Demo	Manual pod delete $\rightarrow$ ArgoCD restores	Recovery in $<5min$	GitOps reliability
End-to-End Dev Cycle	Write spec $\rightarrow$ generate $\rightarrow$ test $\rightarrow$ MR review	Mergeable output with checks pass	Proves gated enforcement

Failure to meet  $\geq 80\%$  of these = iteration needed.

Detailed Specifications

Repo Structure Spec (homelab-ai-stack-gitops/):

```
text
homelab-ai-stack-gitops/
├──
├──
├──
└──
```

```
├─ argocd-root-app.yaml
├─ ollama/
│   └─ values.yaml
│   └─ modelfiles/                # System prompts
│       └─ preload-job.yaml
├─ dify/
│   └─ values.yaml
├─ docs/
│   └─ this-initiation-doc.md      # Copy of this document
│   └─ architecture.md
│   └─ rag-ingestion-guide.md
└─ SPECS/                          # Project-level specs s
    └─ stack-deployment.yaml      # Initial spec for this
```

### Spec Template (for all features; use in /SPECS/):

#### YAML

```
name: Unique Feature Name
description: Natural language overview.
requirements:
  - Bullet list of must-haves.
  - Performance constraints.
success_criteria:
  - Measurable tests (e.g., "LLM review score ≥8/10").
  - Citation requirements.
deliverables:
  - Files/paths produced.
  - Artifacts (e.g., "Dify workflow JSON").
attribution:
  - Required knowledge base sources.
  - External references (e.g., "Rust Book Chapter X").
```

### Initial Spec Example (SPECS/stack-deployment.yaml – for this project itself):

#### YAML

```
name: AI Stack GitOps Deployment
description: Declarative deployment of Ollama + Dify via A
requirements:
  - ArgoCD root app syncs without errors.
  - Ollama preloads specified models with custom prompts.
  - Dify connects to Ollama and enables RAG.
```

**success\_criteria:**

- All success criteria table above met.
- Manual test query in Dify returns cited response.

**deliverables:**

- Fully synced ArgoCD application.
- Accessible Dify UI.

**attribution:**

- NVIDIA GPU Operator docs.
- otwld/ollama-helm and borispolonsky/dify-helm charts.

## System and Instruction Prompts

**Primary Coding Model Prompt** (ollama/modelfiles/coder-  
primary.Modelfile):

text

FROM qwen2.5-coder:72b-instruct-q4\_k\_m

SYSTEM """

You are a rigorous, spec-driven Rust/Python engineer opera  
Rules (enforce always):

1. Reference the exact spec YAML before any code.
  2. Generate comprehensive tests BEFORE implementation.
  3. Cite knowledge base chunks for any non-original pattern
  4. Structure output: Reasoning → Tests → Code → Self-check
  5. Score your confidence 1-10; if <8, suggest improvements
  6. Use structured JSON for final deliverables when request
- """

PARAMETER temperature 0.7

PARAMETER num\_ctx 32768

**Fast Autocomplete Model Prompt** (ollama/modelfiles/coder-  
fast.Modelfile):

text

FROM deepseek-coder-v2:33b-instruct-q5\_k\_m

SYSTEM """

You are a precise, fast coding assistant. Prioritize speed  
Always cite sources if drawing from knowledge base.  
"""

PARAMETER temperature 0.5

**Dify Default Agent Prompt** (set in UI post-deploy):

"You are a spec-enforcement agent. On any task: Retrieve relevant specs/docs first, then plan → code → test → review with citations."

## Supplementing Documentation

### Architecture Summary (for docs/architecture.md):

- Diagram concept: Client (VS Code) ↔ LAN ↔ Ollama (5080 node) ↔ Dify (RAG/Agents) ↔ GitOps (ArgoCD).
- Data flow: Spec written → LLM generates (citing RAG) → Tests run → Review gates merge.

### RAG Ingestion Guide (docs/rag-ingestion-guide.md):

1. Upload sources to Dify Knowledge: Rust book PDF, codebases as folders, ArXiv papers.
2. Settings: nomic-embed-text; hybrid retrieval; rerank enabled.
3. Automation: Future script to sync git repos via Dify API.

## Next Steps to Kick Off

1. Create GitLab repo with above structure; commit Helm values, modelfiles, this doc.
2. Apply ArgoCD root app manifest.
3. Monitor initial sync (model pulls ~1hr).
4. Access Dify UI → configure Ollama models → ingest initial knowledge.
5. Install Continue.dev → point to Ollama endpoint → test.
6. Write first real spec in /SPECS/ → execute workflow.

This document is complete and self-contained—copy it into your repo as the bootstrap artifact. It enforces the spec-driven paradigm from day one. Once deployed, the system itself can refine future specs.

RAG Ingestion Examples

GitLab CI Integration



more concise